情報処理学会論文誌　デジタルプラクティス　Vol.2　No.4　40–49 (Oct. 2021)

ミドルウェア製品開発に対する自動バグ修正技術の適用事例

池田 翔[1,a)]　中野 大扉[1,b)]　亀井 靖高[1,c)]　佐藤 亮介[2,d)]　鵜林 尚靖[1,e)]　久保田 学[3,f)]
矢川 博文[3,g)]　吉武 浩[3,h)]

受付日 2021年1月29日, 再受付日 2021年4月20日/2021年6月14日,
採録日 2021年7月21日

概要：近年，自動バグ修正の有用性が高まっており，既存研究では，自動バグ修正ツールである Prophet を用いて OSS の 69 個のバグのうち 15 件の正しいパッチを生成することに成功している．一方で企業内ソースコードに対して自動バグ修正を適用した場合，実際に期待する形での修正が難しいことも報告されている．自動バグ修正を実際のソフトウェア開発現場で導入するためにはコーディングやテスト工程におけるバグ修正を考慮したうえで，開発現場のプロセスへの導入方法やプロセス改善の必要性について考える必要がある．本稿では，ミドルウェア製品に対して実際に企業（富士通九州ネットワークテクノロジーズ株式会社）で使われている開発プロセスに沿って自動バグ修正を適用し，企業内ソースコードに対する自動バグ修正の有用性や自動バグ修正適用に向けた課題と方針などを 7 つの知見と 2 つの方針として報告する．たとえば，知見としては Prophet で修正可能なバグが全体の 21% であったこと，方針としては単体テストの工程が自動バグ修正ツールに適していることを明らかにした．

A Case Study of Applying Automatic Program Repair Against Middleware Product

Sho Ikeda[1,a)]　Daito Nakano[1,b)]　Yasutaka Kamei[1,c)]　Ryosuke Sato[2,d)]　Naoyasu Ubayashi[1,e)]
Manabu Kubota[3,f)]　Hirofumi Yagawa[3,g)]　Hiroshi Yoshitake[3,h)]

Received: January 29, 2021, Revised: April 20, 2021/June 14, 2021,
Accepted: July 21, 2021

Abstract: The usefulness of automatic program repair has been increasing, and an automatic program repair tool, Prophet, has successfully generated correct patches for 15 out of 69 bugs in OSS. On the other hand, it has been reported that when automatic program repair is applied to proprietary software, it is difficult to fix bugs expectedly. In order to utilize automatic program repair in a software development process, it is necessary to consider how to introduce it, considering the debugging in the testing process, and consider the necessity of process improvement. In this paper, we apply automatic program repair to middleware products according to the development process used in a company (Fujitsu Kyushu Network Technologies Limited) and report the usefulness of automatic program repair for proprietary software, as seven findings and two policies for applying automatic program fixing. We found, for example, that Prophet could fix 21% of the bugs, and that the unit testing process is suitable for automatic program repair tools.

1. はじめに

現在，ソフトウェア開発においてテストの自動化，ビルドの自動化など技術の発展に伴い様々な工程の自動化が進められている [1]．開発の自動化を研究する分野のひとつに，テストの実行履歴を用いてバグの原因箇所を推定し，修正パッチを自動生成する技術（自動バグ修正技術）が積極的に研究されている．開発にかかる時間のうち 25% を占める [2] と言われるバグ修正作業を自動化できれば開発効率は大幅に上昇するため，自動バグ修正は将来において発展が期待される魅力ある研究分野である．

自動バグ修正技術に関して，OSS（Open Source Software）開発プロジェクトや企業プロジェクト [3]-[5] の両方

1 九州大学
Kyushu University, Fukuoka 819–0395, Japan
2 東京大学
The University of Tokyo, Bunkyo, Tokyo 113–0033, Japan
3 富士通九州ネットワークテクノロジーズ株式会社
FUJITSU KYUSHU NETWORK TECHNOLOGIES LIMITED, Fukuoka 814–8588, Japan
a) ikeda@posl.ait.kyushu-u.ac.jp
b) nakano@posl.ait.kyushu-u.ac.jp
c) kamei@ait.kyushu-u.ac.jp
d) rsato@is.s.u-tokyo.ac.jp
e) ubayashi@ait.kyushu-u.ac.jp
f) m-kubota@fujitsu.com
g) yagawa.hirofumi@fujitsu.com
h) yoshitake.hiro@fujitsu.com

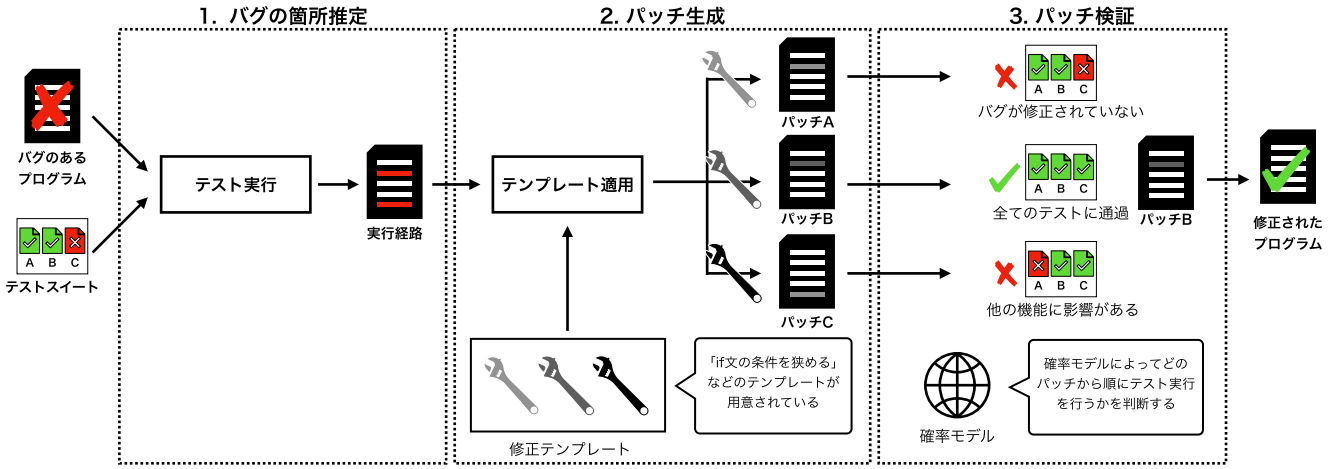


図 1　G&V における自動バグ修正の流れ

Fig. 1　Overview of automatic program repair in the context of generate and validate techniques.

に対する適用事例の報告がある．自動バグ修正ツールの 1 つである Prophet [6] は 8 つの OSS 開発プロジェクトから収集した 69 個のバグに対して修正を試み，15 件の正しいパッチを生成することに成功している．開発者と同等の修正パッチが生成できたことから，修正が有効であることを示している [6]．一方で，実際に期待する形での修正が難しいことも報告されている [3], [4]．

先行事例から，自動バグ修正ツールの修正率は不十分であるものの，我々は今後修正性能の向上などでソフトウェア開発現場での適用が可能になると期待している．しかしながら，自動バグ修正の適用を実際のソフトウェア開発現場で成功させるためには，単に修正性能の向上だけでは難しいと考える．たとえば，開発プロセスに自動バグ修正のプロセスをどう組み込むか，自動バグ修正によってテスト仕様やテストプログラムの書き方は以前の開発プロセスと比較して変化するのか，またテストをどう実施すべきか，さらには，適用コストに対してどの程度の恩恵が得られるか，などの検討・評価を行っていくことが求められる．

本稿では，上記に対する知見を得るために，富士通九州ネットワークテクノロジーズ株式会社（以下 QNET）におけるミドルウェア製品に対して自動バグ修正ツール Prophet を適用し，調査で得られたデータから自動バグ修正ツールの今後の発展とプロセス改善に関する議論を行う[*1]．具体的には，QNET におけるミドルウェア製品から収集した実際のバグに対して自動バグ修正を適用し，その後，今後必要と考えられる自動バグ修正の適用範囲拡大，および，開発プロセスに適した自動バグ修正の導入の指針について検討する．

[*1] 本稿は，本会ソフトウェア工学研究会主催のソフトウェアエンジアリングシンポジウム SES2020 で発表した内容 [7] を改訂したものである．実務者に分かりやすい形に論文構成を見直すとともに，身近に入手可能な参考文献を追加した．

2. 自動バグ修正

本章では自動バグ修正の流れとその適用事例，および本調査で使用する自動バグ修正ツールである Prophet について述べる．

2.1 自動バグ修正の概要

本調査において使用する自動バグ修正ツールは，Gazzola ら [8] がまとめた自動バグ修正技術の分類から選定した．調査対象プロジェクト（詳細は 3 章）では C 言語が採用されているため，分類されているツールの中で C 言語対応であり，かつ，ツールが公開されている，という 2 つの条件を満たす自動バグ修正ツールを探した．この条件を唯一満たしたのは Long らの Prophet [6] のみであったため，本稿では Prophet を使用するツールとして選択した．

Prophet の特徴は，修正の際に利用するテンプレートを用意している点，内部に OSS 開発者の修正を機械学習した確率モデルを保持している点の 2 つである．この 2 点の特徴を，生成と検証を繰り返す手法（Generate and Validate，G&V）の一般的な自動バグ修正ツールの流れとともに説明する．修正の流れの概要は**図 1** のとおりである．

G&V では，ツールの実行にバグのあるプログラムとバグを引き起こすテストを含むテストスイートを必要とする．このテストスイートはテストの自動化がなされていることが必要である．G&V では主にバグの箇所推定，パッチ生成，パッチ検証の 3 つのステップによって自動バグ修正を行う．

バグの箇所推定　テストスイートを実行し，バグを引き起こすテストの実行経路からバグの箇所の推定を行う．このステップではバグの箇所の候補のリストが生成される．

パッチ生成　バグの箇所の候補に対してプログラムを変異させ，パッチを生成する．Prophet は抽象構文木上の一点

表 1　Prophet の修正テンプレート
Table 1　Predefined patch templates in Prophet

テンプレート名	テンプレートの概要
Tighten	if 文の条件式を狭める．
Loosen	if 文の条件式を緩める．
Add Guard	あるステートメントに対してガードをつける．
Insert Guarded Control Flow	あるステートメントの前にガード付きのコントロールフローステートメントを挿入する．
Initialize	メモリの初期化をステートメントの前に挿入する．
Replace	ステートメントの一つの値を別の値に置き換える．
Copy and Replace	あるステートメントをコピーし，別のステートメントの前に挿入後，上記の Replace を行う．

に対して修正テンプレート（**表 1**）を適用することで，1 行程度のパッチを生成する．このステップでは大量のパッチが生成され，次のパッチの検証ステップに移る．

パッチ検証　パッチ生成で生成されたパッチをプログラムに適用し，テストスイートによって検証を行う．最初にバグが検出されていたテストに通過するかだけでなく，パッチ適用前に通過していたテストに失敗しないかを確認する．すべてのテストスイートに通過したパッチが修正パッチとして出力される．Prophet はこのステップで確率モデルを使用し，パッチを検証する順番を変更している．この確率モデルにパッチを入力として与えると学習した開発者のパッチとの類似度が出力される．検証を行う際はこの類似度が高いパッチから評価することで，開発者の修正に近いパッチを生成することが可能になっている．

Prophet を利用する際のプロセス　最後に，本稿で想定するソフトウェア開発者が Prophet を利用する際のプロセスを示す．

(1) ソフトウェア開発者がソースコードを作成する．
(2) ソフトウェア開発者は作成したソースコードをもとに，単体テストを作成，および，実施する．
(3) ソフトウェア開発者は単体テストでバグが検出された場合，該当ソースコード，単体テストを Prophet に入力する．
(4) Prophet によって修正パッチが生成された場合，ソフトウェア開発者はそのパッチの可否を判断する．
(5) パッチが受け入れられる場合，ソフトウェア開発者はパッチを採用する．
(6) パッチが受け入れられない場合，ソフトウェア開発者はバグの修正を行う．この際，Prophet のパッチを参考にする．
(7) ソースコード作成，および，テスト実施を終了する．

2.2　自動バグ修正と適用事例

自動バグ修正ツールを提案している論文の多くは，OSS にツールを適用し，その評価を行っている．一方で，実際に企業内で開発されたソースコードに対しての適用事例は少ない．

内藤ら [3] は，Java で記述された企業内ソースコードに対して GenProg [9] を含む複数の自動バグ修正手法を適用し，修正率は 7.7%（=1 件/13 件）であった．その結果から得られた自動バグ修正の実用化の障壁について議論している．

Noda ら [4] は，Java で記述された 150 を超える企業内ソースコードから得られたバグを用いて自動バグ修正ツール Elixir [10] を適用し，修正率は 10.0%（=2 件/20 件）であった．また，その結果からテストケースの不足やツールの成功率の低さを指摘している．

本調査では自動バグ修正の企業内ソースコードへの適用事例のひとつとして，適用する際に得られた知見について報告する．

3.　適用対象プロジェクト

対象プロジェクトの概要　本稿での対象プロジェクトは C 言語で開発されたミドルウェア製品である．このミドルウェア製品は母体にあたる部分の開発はすでに終了しており，複数回の派生開発によって機能追加を行っている．母体に当たる部分は約 16 kLOC であり，派生開発では母体の一部を含み約 7.2 kLOC である．ウォーターフォール型の開発が行われ，派生開発での開発人数は平均 3 人である．

品質保証に関する指針　対象プロジェクトはバージョン管理システムによって管理されており，派生開発後の総コミット数は 759 である．開発中のコミットのポリシーとしては，コンパイルエラーが起きない，プロジェクトのビルド・単体テストが通過するなどがある．なお，単体テスト作成時，分岐網羅率（branch coverage，C1）が 80% 程度を達成するよう，テスト項目が作成されている．

4.　適用調査

4.1　調査観点

今回，3 章で述べたミドルウェア製品を対象に Prophet による自動修正を試みた．Prophet はテストスイートによってパッチの検証を行うアプローチを採用しているため，バグが直ったかどうかを確認するためのテストコードが必要となる．そのため，今回の調査では QNET におけるミドルウェア製品開発プロセス（開発標準および品質指標）に沿って実際にテストコードを作成し，開発現場での自動バグ修正の適用可否を評価した．評価の観点は以下の 3 つである．

（観点 1）自動バグ修正技術の適用可能性　現状，自動バ

グ修正が適用可能なバグの割合がどの程度かを調査した．

（観点 2）自動バグ修正の性能　自動バグ修正を適用し，どの程度修正可能かを調査した．

（観点 3）自動バグ修正の適用プロセスとコスト　現在のミドルウェア製品開発プロセスにおいて自動バグ修正を適用するコストをテスト作成の観点から調査した．

4.2 （観点 1）自動バグ修正技術の適用可能性

動機　本稿で適用する自動バグ修正ツール Prophet は，先行研究 [6] によって高い性能が報告されているものの，修正可能なバグが 1 ファイル内の 1 行程度という制約がある．そのため，複数ファイルかつ複雑な修正が予想される実製品レベルのバグを修正するのは困難であると我々は考える．この制約がどの程度現実的かについて調査・考察することで，今後の自動バグ修正に役立つ知見や方向性を示す．

方法　対象プロジェクトのリポジトリ内からバグ修正をしているコミットを抽出し，Prophet によって修正できる可能性があるかを調査した．具体的には，以下の手順（1）から手順（3）で調査対象コミットを抽出した．その後，手順（4）と手順（5）でコミットを修正ファイル数，修正行数，Prophet によって修正できる可能性があるか，の 3 項目で分類した．

(1) 対象プロジェクトの全コミット履歴からコミットメッセージを抽出
(2)「バグ」や「修正」などのキーワードを含むコミットを抽出
(3) 手順（2）で抽出されたコミットのコミットメッセージを目視調査し，バグ修正を行っていると判断したコミット（バグ修正コミット）を抽出
(4) バグ修正コミットによって追加・削除された差分行数，差分ファイル数を調査
(5) 差分行数が単一行かつ単一ファイルの編集であり，開発者の修正が Prophet のテンプレートに含まれているものを適用対象コミットとした

調査結果　全コミット 759 件のうち，コミットメッセージにキーワードが含まれるものが 130 件であった（手順 (2)）．そのうち，コミットメッセージからバグ修正を行っていると判断できたコミットは 21 件であった．21 件のコミットに対して，実際にバグ修正を行っている行を特定できたものは 14 件であった（手順 (4)）．差分からバグが特定できなかった 7 件のコミットに関しては，差分の行数が多くコミットメッセージのみからはバグの箇所が判断できないものや，コミットメッセージが「バグ修正」のみなどのコミットがあげられる．

14 件のコミットを修正ファイル数，修正行数，Prophet によって修正できる可能性の 3 項目によって分類した（**表 2**）．その結果，Prophet によって修正できる可能性があるコミットは 3 件であった．

表 2　バグ修正をしていると判断したコミットの内訳
Table 2　A breakdown of the commits that are classified as bug fixes.

修正ファイル	修正行	Prophet で修正可能	個数
複数ファイル	複数行	不能	2
単一ファイル	複数行	不能	5
	単一行	不能	4
		可能	3

プログラム 1　Commit 1 プログラムの概要と開発者の修正

```
1          void debug(){
2              int config_val = read_config();
3              char *data = malloc(config_val);
4              // config_valの値でメモリ確保
5              ...
6  -           for(i = 0; i < 256; i++){
7  +           for(i = 0; i < config_val; i++){
8                  printf("%d", data[i]);
9              }
10         }
11         int read_config(){
12             int config_val = 256;
13             ...
14             // 設定ファイルの値で
                   config_valを変更
15             ...
16             return config_val;
17         }
```

（知見 1）適用対象の割合

ミドルウェア製品のバグのうち，Prophet の適用対象となったのはバグの全体の 21%（=3/14）であった．21% という数字をどう捉えるかは判断が難しく，21% しか適用できないと考えるか，それとも自動バグ修正というある意味で夢の技術の適用可能性が現時点でも 21% もあると考えるかにより評価は大きく異なる．

適用対象のバグ修正コミット　以降，3 件の適用対象コミットをそれぞれ Commit 1，Commit 2，Commit 3 と呼ぶ．Commit 1 は設定ファイルに関係するバグ，Commit 2 は NULL チェック，Commit 3 は無限ループのバグである．各適用対象コミットについてバグの内容と開発者の修正について説明する．

Commit 1（**プログラム 1**）はデバッグ用の関数に関する修正である．プログラムでは，設定ファイルから値を読み込み，その値によってメモリを確保している（2，3 行目）．プログラム中に for 文のループ変数に応じてメモリを参照する箇所がある．メモリを確保した以上に `for` 文内の処理が実行された場合，範囲外にアクセスしてしまいコアダンプが起きる，という内容である．開発者の修正は `for` 文のループ回数を設定ファイルの値によって制限し，コアダンプを回避している．

Commit 2（**プログラム 2**）では，関数ポインタ `handler` が `NULL` だった場合，`handler(args)` でコアダン

プログラム 2　Commit 2 開発者の修正

```
1  -        handler(args);
2  +        if(handler != NULL){
3  +            handler(args);
4  +        }
```

プログラム 3　Commit 3 開発者の修正

```
1           while(1){
2               ...
3               else{
4  -                continue;
5  +                break;
6               }
7           }
```

プを出力するいう内容である．開発者は handler が NULL でない場合のみ handler(args) を実行するように修正している．

Commit 3（**プログラム 3**）では，else 文に入った場合無限ループが発生しており，break 文を挿入することで修正している．

適用対象コミットの 3 件はすべてバグではあるものの，修正という観点では難易度は低い．適用可能範囲は狭いが自動バグ修正を適用できる可能性があると言える．

（知見 2）修正の容易性

実製品のバグは修正が困難なものが多く，自動バグ修正で対処できるものではないという先入観が我々にはあったが，今回の調査で自動バグ修正でも対応可能なものが少なくないことが分かった．要求仕様に起因するバグや設計の見直しが必要なバグについては，現時点の自動バグ修正では確かに対処が困難である．しかし，今回の調査のようなバグはコーディングミスやテスト漏れに近く，自動バグ修正で十分対応可能である．前者のバグ修正は開発者自身の手で，後者は自動バグ修正で行うなどの分業が期待される．

4.3 （観点 2）自動バグ修正の性能

動機　自動バグ修正ツールは特定の範囲の対象に対しては高い修正性能を示している一方で，実際には期待どおりの修正性能が発揮されない場合も報告されている．現状のツールでどの程度修正が可能であるかを調査した．

方法　調査した 3 件に対して，Prophet を適用した．

文献 [8] では，自動バグ修正においてテストスイートにすべて通るだけのパッチでなく，開発者に受け入れられるパッチを生成することが求められると述べられている．そこで，得ることができたパッチについて，以下の点を評価した．

(1) パッチ内で修正された行は，（観点 1）で特定したバグのある行か

(2) 修正されたコードは開発者の修正コードと同様の動きをするか

(3) 開発者に受け入れられるか

各項目について九州大学・QNET の共同ミーティング内で議論し，判定した．

表 3　作成したテスト

Table 3　Test cases that we implemented.

コミット ID	テストの個数	対象関数の LOC	作業時間（分）
Commit 1	2	1,397	300
Commit 2	3	41	120
Commit 3	2	35	50

単体テストの作成　Commit 1，2，3 のバグに Prophet を適用するのに必要なテストを作成した．Commit 1，2，3 のバグはいずれも軽微であり，修正も容易である．そのため，開発者がテストを作成せずにコミットを行い，単体テストが作成されていなかった．このように適用対象バグに対してバグを発現させるテストが存在しない事例は，先行研究でも報告されている [3], [4]．今回の調査では，単体テスト時に自動バグ修正を適用するという想定の元で，筆頭著者が実際にミドルウェア製品の該当ソースコード箇所を読み，単体テストを作成した．その際，ソースコード理解からテスト作成完了までにかかった作業時間の合計を記録した．ソースコード理解とテスト作成の作業を完全に分離できなかったため，それぞれの作業時間ではなく作業時間の合計のみを記録した．**表 3** にその概要を示す．ここで，Commit 1 の対象関数の LOC が他のコミットに比べて非常に大きくなっているが，これは対象関数が多数の分岐を持つ switch 文からなっているためであり，着目すべき箇所の LOC は他のコミットと同程度である．

テスト内容について　単体テストの作成にあたっては，以下のテスト内容に示すようにコード網羅率を高めることに留意した．一般的に網羅率を向上させると自動バグ修正の性能が高まるためである [11]．

Commit 1　設定ファイルの値について，for 文のループ回数と等しくバグが発生しない 256，for 文のループ回数より小さいためバグを発生させる 256 より小さい値，の 2 つのテストを用意した．

Commit 2　関数の引数（handler を含む構造体）が NULL でないかを確認するテスト，handler が NULL でないテスト，handler が NULL でバグを引き起こすテスト，以上 3 つのテストを作成した．

Commit 3　else 文に入るテスト，入らないテストの 2 つのテストを作成した．

調査結果　適用結果は**表 4** のとおりである．3 件の調査対象コミットのすべてについて，Prophet によってパッチが生成された．現実のミドルウェア製品においても，自動バグ修正によってパッチが生成可能なバグは存在することが

表 4　修正パッチの評価
Table 4　Evaluation of generated patches.

コミット ID	パッチ生成	バグ同定	プログラムの動き	開発者に受け入れられるか
Commit 1	成功	失敗	異なる	不適切
Commit 2	成功	成功	同じ	適切
Commit 3	成功	成功	同じ	適切

表 5　正答パッチ率の先行研究との比較
Table 5　Comparison of the number of correct patches with previous studies.

	対象バグ数	正答パッチ数	正答パッチ率
本研究	14	2	14.3%
内藤ら [3]	13	1	7.7%
Noda ら [4]	20	2	10.0%

分かった．

3 件の調査対象コミットのうち，2 件のパッチについて開発者に受け入れられるパッチであると判断した．（詳細は知見 3 後の“受け入れられるパッチ”を参照されたい．）内藤ら [3] の調査では 13 件のバグに対して実験を行い，1 件のバグについて開発者と同等のパッチを生成することができたと述べている．Noda ら [4] の調査では 20 件のバグに対して実験を行い，2 件のバグについて開発者と同等のパッチを生成することができたと述べている．本稿では 14 件のバグのうち 2 件のバグについて開発者と同等のパッチが生成できたため，2 つの先行事例を補強する結果になっている．内藤らの正答パッチ率は 7.7%（=1/13），Noda らの正答パッチ率は 10.0%（=2/20），今回の正答パッチ率は 14.3%（=2/14）である（**表 5**）．件数自体が少ないため数値のブレはあるが，言語や使用ツールにかかわらず現状では 10% 前後くらいしか正しいパッチは得られないと考えられる．

Prophet を提案している論文 [6] では，正答パッチ率は 21.7%（=15/69）と報告されており，内藤ら，および，Noda らの調査より高い正答パッチ率となっている．しかし，これは調査対象のプロジェクトを Prophet の修正可能範囲にあるコミットを十分に含むものに限定しているためであると考えられる．

（知見 3）パッチ生成の可能性
適用可能範囲は限定的であるものの，対象のバグに関しては実製品のバグにおいても開発者と同等のパッチを生成することが可能である．ただし，正しいパッチが得られるのはバグ全体のうち 10% 程度であり，現時点では，開発現場の期待に応えることは難しい．

各調査対象コミットについて生成されたパッチの詳細と評価について述べる．

プログラム 4　Commit 2 開発者の修正と生成されたパッチ

```
1          // 開発者の修正
2  -       handler(args);
3  +       if(handler != NULL){
4  +           handler(args);
5  +       }
6          // 生成されたパッチ
7  -       handler(args);
8  +       if(!(handler == 0)){
9  +           handler(args);
10 +       }
```

プログラム 5　Commit 3 開発者の修正と生成されたパッチ

```
1          // 開発者の修正
2          while(1){
3              ...
4              else{
5  -               continue;
6  +               break;
7              }
8          }
9          // 生成されたパッチ
10         while(1){
11             ...
12             else{
13 +               if(1)
14 +                   break;
15                 continue;
16             }
17         }
```

受け入れられるパッチ　Commit 2 に対して Prophet が生成したパッチ（**プログラム 4**）は `handler` のアドレスが 0，つまり `NULL` でないときのみ `handler(args)` を実行するように修正している．このパッチは元のプログラムの機能を維持し，バグも修正しているため，最善の方法ではないものの，開発者に受け入れられるパッチであると判断した．

Commit 3 に対して Prophet が生成したパッチ（**プログラム 5**）は `break` を挿入して無限ループを解消している．`continue` を削除していないため開発者の修正と完全に同じ修正ではなく，直接パッチとしてソースコードに適用することはできないが，開発者に受け入れられるパッチだと判断した．開発者とのパッチの差は差分行数で考えると大きくないため，開発工程においてはコードレビューで対応可能であると考える．

（知見 4）パッチレビューの必要性
生成した修正パッチをソースコードにマージするためには，テストにすべて通過したかのみで判断するのではなく，通常のコードレビューと同様にパッチのレビューを行う必要がある．

受け入れられないパッチ　Commit 1 に対して Prophet が生成したパッチ（**プログラム 6**）は，設定ファイルを読み込んでいる関数に対してのパッチである．設定ファイルを読み込む箇所を `scanf` に変更し，実質，設定ファイル読み込みを無効化している．この `scanf` への入力は通常の

プログラム 6　Commit 1 生成されたパッチ

```
1          // 生成されたパッチ
2          int read_config(){
3              int config_val = 256;
4              ...
5  -           fgets(line, n, file);
6  +           scanf(line, n, file);
7              ...
8              return config_val;
9          }
```

使用法から逸脱したものではあるが，実際にコンパイル可能なものであり，今回の実験環境では実質何もしない関数呼び出しとなっている．設定ファイルを読み込む関数は `config_var` を `for` 文の実行回数と同じ値で初期化しており，この初期値によってメモリが確保された場合，コアダンプは発生しない．

このパッチはコアダンプを回避しているためバグを修正しているとも言えるが，設定ファイルを読み込むという本来の機能を維持していない．また，環境に依存するコードでもあるため，開発者に受け入れられないパッチと判断した．このパッチが生成されたのは，自動バグ修正ツールに与えるテストケースが不十分であったためと考えられる．たとえば，設定ファイルの値を読み込んでいるかを確認するテストをテストケースに追加されれば，受け入れられるパッチが生成される可能性がある．

このパッチは開発者には受け入れられないものの，設定ファイルを読み込む周辺でバグが発生していることが分かる．開発者がバグ修正を行う際にこの情報を利用することでデバッグ作業に有用な情報になる可能性がある．従来のバグ検出技術やツールと比較して，実際にどの程度有用であるかの調査・比較は今後の課題とする．

（知見 5）自動バグ修正の付帯的効果
開発者と同等の修正パッチが得られない場合でもバグ同定などの情報からデバッグ作業に役立つ情報を利用できる場合がある．

4.4 （観点 3）自動バグ修正の適用プロセスとコスト

動機　自動バグ修正の導入を実際のソフトウェア開発現場で成功させるためには，単に修正性能が高いだけでなく，現場の開発プロセスにどう組み込むか，テスト仕様やテストプログラムは現状のまま適用できるのか，開発プロセスの改善は必要か，などについて考える必要がある．さらに，実施にどの程度のコストがかかるかについても調査する必要があると考える．

ここでは，実際のミドルウェア製品開発プロセスにおいて作成されるテスト仕様で自動バグ修正が適用可能かを調査し，その後，自動バグ修正を適用するコストについてテスト作成の観点から調査した．

方法　4.3 節において作成したテストについて，テスト内容や追加するべきテスト等についてミドルウェア製品開発プロセスでの基準と比較を行う．比較項目はテストの種類，および，分岐網羅率（C1）である．本調査で作成したテストとミドルウェア製品開発プロセスで作成されるテストを比較することで，開発プロセスで作成されるテストをそのまま自動バグ修正で利用可能かを調査した．本調査では，テストを作成する際，バグ混入時のコミットメッセージを参考にした．

表 6　作成したテストの比較
Table 6　Evaluation of test cases that we implemented.

コミット ID	テスト個数	対象関数の LOC	分岐網羅率 (C1)
Commit 1	2	1397	低
Commit 2	3	41	高
Commit 3	2	35	高

調査結果　作成したテストと品質指標との比較結果は**表 6** のとおりである．

テスト内容について　はじめに，今回作成したテスト内容の妥当性について述べる．今回筆頭著者が作成したテストをプロジェクトに精通した著者が内容を確認し，通常の開発プロセスで作成される可能性が十分にあることを確認した．また，これらの単体テストは Prophet の入力になるための条件も満たしている．

知見 3 でも述べたが，自動バグ修正を適用するには網羅率を上げるのが望ましい．ただ，今回作成した Commit 1 に対する単体テストは結果として網羅率が十分ではなく，テスト内容も不足している．不足しているテストの例として，境界値の前後のテストだけでなく境界値のテストや設定ファイルの種類を増やすべきである．一方，Commit 2 に関しては調査で作成したバグを引き起こすテストは開発時にも用意されているべきであり，開発時のテスト項目の漏れが考えられる．これらの問題を解決するための方法として，まず，自動バグ修正を利用するために必要な最低限の単体テスト，すなわち，ある程度の網羅率を担保するテストを始めに用意する．次に，これに加えて品質保証として不足しているテストを開発者が新たに追加する．この手順を取ることによって，通常の開発プロセスの単体テストをより拡充したものを作成可能であると考えられる．

（知見 6）品質保証からの視点
自動バグ修正を適用するためのみに作成した最小限のテストでは品質保証のテストとしては不十分であるが，作成したテストの一部は品質保証のテストのための資産に追加できる．自動バグ修正を適用するために作成したテストを既存の単体テストスイートに随時追加して行くことが望まれる．このプロセス改善のハードルは低いので，開発現場では容易に実現できると思われる．

実施のコストについて　表3の作業時間は筆頭著者がソースコードを読み，ソースコードのコンパイルやテストを作成した時間の合計である．テスト作成はCommit 1，Commit 2，Commit 3の順に行ったため時間が減少している．また，Prophetの1回の実行時間は5分程度であり，Prophetを使ったことがないユーザでも手順書に従えば1時間程度で動かせるようになる．

（知見7）導入のコスト
ミドルウェア製品への理解が浅い作成者でもテストの作成方法について3件程学習することで，作業時間を単体テストレベルで1時間程度の時間に抑えることができる．学習による時間削減の理由としては，テスト作成による慣れが大きい．実際の製品開発では，開発者自身が単体テストを作成するので，時間は1時間より大幅に短くなることが予想され，コスト面での負荷はあまりないと考えられる．

5. 今後の自動バグ修正に向けた方針

ここでは調査から得られた知見から今後の自動バグ修正に向けた方針を考える．

5.1 知見に対する意見

本節では九州大学とQNETの共同ミーティングの議論で得られた自動バグ修正適用の知見（特に，開発現場に対する適用の一事例としての知見や意見）についてまとめる．各項目の評価は以下のとおりである．○現状に満足できる．△改良が必要である．×大幅な改良が必要．

×（**知見1**）**適用対象の割合**　現時点では対応できるバグの割合が小さく，導入の効果が薄い．将来的に対応可能なバグが増えることを期待する．

△（**知見2**）**修正の容易性**　軽微なバグは修正の難易度は高くはないが，開発の工程前半で数多く発生するものであり，自動修正による効率向上に期待ができる．

△（**知見3**）**パッチ生成の可能性**　知見1でもそうだが，現状では適用範囲が狭いため評価が難しい．今後適用範囲が広がり，パッチも生成できれば嬉しい．また，現時点で開発者と同等のパッチが生成できたことは評価できるが，3件中2件は例としての数が少ない．さらなる調査が必要と考える．

△（**知見4**）**パッチレビューの必要性**　通常のコードレビュープロセスと同等に行う必要があるため，一部プロセスの改善を行う必要がある．バグ修正作業がどの程度減るかはまだ不明であるが，その時間をコードレビューに充てることができると考える．

△（**知見5**）**自動バグ修正の付帯的効果**　開発者と同等のパッチが生成できることが望ましいが，バグ同定でも開発者の負担を減らすことはできる．パッチ生成に成功したかどうかだけでなく，修正パッチとは別にバグ同定結果が分かりやすく開発者に通知できると嬉しい．

○（**知見6**）**品質保証からの視点**　単体テストを作成する過程で自動バグ修正が適用できると望ましい．

○（**知見7**）**導入のコスト**　1時間より大幅に短い時間であれば，通常の単体テスト作成時間内で自動バグ修正を追加で利用できるので魅力的である．

5.2 自動バグ修正の適用範囲

知見1，2，および，3より，自動バグ修正ツールが対応可能なバグの割合は開発現場の期待水準には達していない．本節では，自動バグ修正ツールの適用範囲を拡大する際の方向性を議論する．

適用調査では既存のProphetのテンプレートでどの程度修正が期待できるかの調査を行ったが，バグの箇所が特定できた14件のうち3件のみが理論上修正可能であった．上記の3件を除いた11件中4件は単一ファイル・単一行の修正であり，バグの内容を確認したところ，テンプレートの追加によって自動バグ修正を適用することができる可能性があることが分かった．

単一ファイル・複数行のバグの中には連続する複数ステートメントをif文によってガードする修正が2件存在した．このようなバグに対してはProphetにテンプレートを追加するだけでは対応が難しい．今後は連続した複数行を対象とした自動バグ修正ツールの開発やProphetのAdd Guardテンプレートを複数行に拡張する，などの対応が必要である．

上記で述べたテンプレートの追加および拡張を行うことで，現在の14件中3件（約21%）から14件中9件（約64%）のバグが将来的に修正可能になると考えられる．残った5件については修正行数が多く，関数追加やリファクタリングが行われており自動で修正することは難しい．これらのバグに対しては人手での修正や別プロセスで工夫してバグの混入を防ぐなどの対策を考える必要がある．

自動バグ修正適用に向けた方針1
自動バグ修正ツールの適用範囲は限定的であるため，ツールをそのまま使うのではなくテンプレートの追加，もしくは複数行への対応を検討すると適用範囲を広げられる．

5.3 開発工程とコスト

知見6および7より単体テストが自動バグ修正で利用可能なこと，知見4より導入の際に考慮するべきこと，知見5より開発プロセスにおける自動バグ修正の利点が分かった．本節では**表7**を参考に，コストやProphetの特性から開発プロセスへの組み込みについて議論する．表7ではミドルウェア製品開発における各工程をテスト作成，実

表 7　現在のミドルウェア製品開発工程におけるコスト
Table 7　Costs during development process in the context of our target middleware product projects.

工程	テスト作成	テスト実施	問題特定	問題修正	総コスト
コーディング	-	-	低	低	低
単体テスト	高（物量）	低〜中（物量）	低〜中	低〜中	中（物量）
結合テスト	高（難易度）	中（実行時間）	中〜高（難易度）	中〜高	高（難易度）
システムテスト	高（難易度）	中（実行時間）	特高（難易度）	特高（アーキ[1]）	特高（難易度）

[1] アーキテクチャレベルの問題

施，問題特定，修正，総コストに細分化し，低，中，高，特高の 4 段階で評価した．各コストは，各工程にかかる金額，人件費，時間などを考慮し，絶対評価を行った．たとえば，結合テストではテストを作成するにあたって，テストの事前条件を作り出す必要があるためテスト作成のコストを高としている．また，コーディングの工程ではテスト作成のコストを低としているが，これは厳密なテスト作成ではなく動作確認やデバッグのことを指している．

Prophet は 1 行程度の修正のみ可能であるため，問題特定や問題修正がそれほど困難でない単体テスト工程が導入に最適な工程であると考えられる．また，前述したように，今回行った調査では自動バグ修正にかかる時間は 5 分程度であったため，開発者が単体テストを実行する際に追加で自動バグ修正を適用しても時間の観点で大きなコストにはならない．

単体テスト工程に導入する場合，ミドルウェア製品の特性からパラメータ分岐が多いため，テスト作成工程のコストを考慮する必要がある．テスト作成のコスト改善には自動テスト生成などの技術を用いることが可能である．たとえば，今回の Commit 2 において `handler` が `NULL` の場合のテストが抜けていたような事例では，自動テスト生成によって handler が NULL でない場合のテストを作成し，開発者自身で NULL の場合のテストを作成することで自動バグ修正をより効率よく利用することができると考えられる．以上からプロセス改善につなげるためにはテストの自動生成などと併用して自動バグ修正を導入するのが効果的であると考える．

自動バグ修正適用に向けた方針 2
自動バグ修正ツールを導入するのは単体テスト工程が最適と考えられる．単体テスト工程はテスト作成にかかるコストが大きいため，テスト自動生成などと併用することでプロセス改善に取り組める．

6. おわりに

本稿ではミドルウェア製品開発プロセスから抽出した実際のバグ 14 件のうち，3 件のバグに対して自動バグ修正ツール Prophet を適用した．その結果，2 件の開発者と同等のパッチ生成に成功し，Prophet によって修正可能な実製品のバグが存在することを明らかにした．ミドルウェア製品開発プロセスにおいて現在の自動バグ修正ツールの適用範囲は 21% と小さく，適用範囲を拡大するために修正テンプレートの追加などが必要である．また，適用調査でのデータから得られた知見に対する企業からの意見をまとめ，単体テストの工程が自動バグ修正ツールに適していることを述べた．

参考文献

[1] 正野勇嗣：ソフトウェア開発自動化の歴史，株式会社マイナビ（オンライン），入手先〈https://news.mynavi.jp/itsearch/article/devsoft/4095〉（参照 2021-01-07）．

[2] Britton, T., Jeng, L., Carver, G. and Cheak, P.: Reversible Debugging Software “Quantify the time and cost saved using reversible debuggers” (2013).

[3] 内藤圭吾，谷門照斗，柗本真佑，肥後芳樹，楠本真二，切貫弘之，倉林利行，丹野治門：企業のソフトウェア開発に対する自動プログラム修正技術適用の試み，ソフトウェアエンジニアリングシンポジウム 2018 論文集，pp.139–147 (2018).

[4] Noda, K. Nemoto, Y., Hotta, K., Tanida, H. and Kikuchi, S.: Experience Report: How Effective is Automated Program Repair for Industrial Software?, *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering* (*SANER*), pp.612–616 (2020).

[5] 富士通研究所：DX 時代のソフトウェア開発を革新 AI によるバグ修正技術を日米で開発，富士通研究所（オンライン），入手先〈https://www.fujitsu.com/jp/group/labs/about/resources/article/202007-aibug.html〉（参照 2020-12-15）．

[6] Long, F. and Rinard, M.: Automatic patch generation by learning correct code, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16*, pp.298–312 (2016).

[7] 池田　翔，中野大扉，亀井靖高，佐藤亮介，鵜林尚靖，久保田学，矢川博文，吉武　浩：ミドルウェア製品開発への自動バグ修正技術適用の試み，ソフトウェアエンジニアリングシンポジウム 2020 論文集，pp.79–87 (2020).

[8] Gazzola, L., Micucci, D. and Mariani, L.: Automatic Software Repair: A Survey, *IEEE Transactions on Software Engineering*, pp.34–67 (2019).

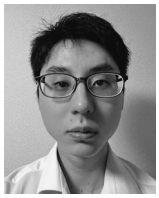
[9] Weimer, W., Nguyen, T., Le Goues, C. and Forrest, S.: Automatically finding patches using genetic programming, *Proceedings of the 31st International Conference on Software Engineering, ICSE '09*, pp.364–374 (2009).

[10] Saha, R. K., Lyu, Y., Yoshida, H. and Prasad, M. R.: Elixir: Effective object-oriented program repair, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (*ASE*), pp.648–659

(2017).
[11] Yi, J., Tan, S., H., Mechtaev, S., Böhme, M. and Roychoudhury, A.: A Correlation Study between Automated Program Repair and Test-Suite Metrics, *Proceedings of the 40th International Conference on Software Engineering*, p.24 (2018).

池田 翔

2019 年九州大学工学部電気情報工学科卒業．2021 年同大学大学院システム情報科学府情報知能工学専攻修了．自動バグ修正の研究に従事．

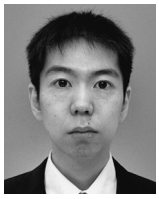


中野 大扉

2016 年九州大学工学部電気情報工学科卒業．2018 年同大学大学院システム情報科学府情報知能工学専攻修了．マイニングソフトウェアリポジトリの研究に従事．

亀井 靖高

2005 年関西大学総合情報学部卒業．2009 年奈良先端科学技術大学院大学情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員（PD）．2010 年カナダ Queen's 大学博士研究員．2011 年九州大学大学院システム情報科学研究院助教．2015 年同大学同研究院准教授．博士（工学）．マイニングソフトウェアリポジトリ，ソフトウェアメトリクスの研究に従事．ESEM 2007 Best Paper Award，MSR 2014 Distinguished Paper Award，2015 年度情報処理学会論文賞など各賞受賞．ACM，ソフトウェア科学会，電子情報通信学会各会員．IEEE Senior Member.



佐藤 亮介

2013 年東北大学大学院情報科学研究科博士課程後期 3 年の課程修了．同年東京大学大学院情報理工学研究科特任研究員．2017 年九州大学大学院システム情報科学研究院助教．2020 年東京大学大学院情報理工学系研究科助教．ソフトウェアの形式的検証の研究に従事．

鵜林 尚靖

1982 年広島大学理学部数学科卒業．1999 年東京大学大学院総合文化研究科広域科学専攻広域システム科学系博士課程修了．博士（学術）．1982〜2003 年（株）東芝に勤務．2003 年九州工業大学情報工学部助教授，2010 年九州大学大学院システム情報科学研究院教授，現在に至る．2003 年度情報処理学会山下記念研究賞受賞．ソフトウェア工学，プログラミング言語モデルの研究に従事．日本ソフトウェア科学会，電子情報通信学会，ACM，IEEE-CS 各会員．本会フェロー．

久保田 学

1994 年九州芸術工科大学 芸術工学部 音響設計学科卒業．1994 年アルパイン株式会社入社．2001 年富士通九州ディジタルテクノロジ株式会社（現富士通九州ネットワークテクノロジーズ株式会社）入社．同社技術戦略本部第二統括部第一テクノロジー部長（現職）．メディア処理技術の研究開発に従事．プロジェクトマネジメント学会会員．

矢川 博文

1988 年長崎大学工学部 電気工学科卒業．1988 年富士通九州ディジタルテクノロジ株式会社（現富士通九州ネットワークテクノロジーズ株式会社）入社．通信キャリア向けネットワークシステム用装置開発に従事．2020 年マツダ株式会社出向，Connected vehicle システムの保守運用業務に従事．

吉武 浩

1991 年九州大学理学部数学科卒業．1991 年富士通九州通信システム株式会社（現富士通九州ネットワークテクノロジーズ株式会社）入社．同社技術戦略本部第二統括部長（現在）．ネットワーク関連システムの研究開発に従事．電子情報通信学会各会員．